Mixed-Model Parallel Implementation of Chimera Grid Assembly

Nathan C. Prewitt[1]
Mississippi State University
U.S. Army Engineer Research and Development Center
Major Shared Resource Center
Vicksburg, MS

Daniel Duffy[2]
Computer Sciences Corporation
U.S. Army Engineer Research and Development Center
Major Shared Resource Center
Vicksburg, MS

Wei Shyy[3]
University of Florida
Gainesville, FL

[1]PET Onsite CFD Lead, DoD ERDC MSRC
[2]Computational Scientist, Computational Sciences and Engineering, DoD ERDC MSRC
[3]Head of the Department of Aerospace Engineering, Mechanics and Engineering Science

Abstract

Improvements in the parallel implementation of the grid assembly function within the Beggar code are presented. A mixed programming model is used combining message passing with shared-memory programming constructs using standard POSIX calls. The POSIX calls are used to store large, common data structures in shared memory. The associated reduction in the total memory requirements allows more processors to be utilized. The effective utilization of this increased processor count is based on improvements in load balancing because of a finer decomposition of the work associated with the grid assembly function. Parallel efficiency is demonstrated using a three store, ripple release, trajectory calculation.

Introduction

The Chimera grid scheme [1] eases the burden of generating multiblock structured grids for geometrically complex domains while retaining the accuracy and efficiency of structured grid flow solvers. This is accomplished by generating structured grids about separate components of the geometry in an independent fashion while relaxing the requirement of block face matching. Instead, the independent grids are allowed to overlap; in fact, they are required to overlap so as to facilitate intergrid communication. This communication between the grids is established via interpolation of the flow solution from the interior of the grids onto the boundaries of any overlapping grids, thus supplying the required boundary conditions.

The independent nature of the overlapping grids makes the Chimera scheme particularly attractive for the solution of aerodynamically driven, moving-body problems. Thus, Chimera has seen application in store separation [2], tilt-rotor aircraft aerodynamics [3], and other interesting areas.

The freedom gained in grid generation comes in exchange for the problem of establishing explicit communication links between overlapping grids. The process of establishing these communication links (hereafter called the grid assembly process) can be rather expensive. The expense increases further with the time-accurate solution of moving-body problems because the grid assembly has to be updated after each time-step of the solution.

Numerous attempts have been made to create robust and efficient algorithms for handling the grid assembly problem [4, 5, 7, 6]. The number of algorithms that have been applied to dynamic, moving-body problems is somewhat less [8, 7, 9]. The number of parallel implementations for dynamic problems is fewer still [10, 11, 12].

Meakin et al. [11, 12, 13, 14] have performed the parallel solution of many dynamic problems using Chimera methods. However, the decomposition of the work related to the grid assembly function is based on the same decomposition used for balancing the work of the flow solver. Modifications in the work distribution, made to improve the load balance of the grid assembly, only lessened the parallel performance of the flow solver, and thus, were detrimental to the parallel performance of the overall simulation [12].

Prewitt, Belk, and Shyy [15] performed the first parallel, Chimera CFD solution of a dynamic problem that used a data decomposition to load balance the grid assembly separate from that of the flow solver. Separate mappings of the grids to processors were used for the grid assembly and the flow solver. However, large load imbalances were still seen in the grid assembly function,

and the relatively small number of grids used in the test problem did not offer much opportunity for scalability.

In order to improve scalability, Prewitt [16] used a finer grain decomposition of the work associated with the hole-cutting part of the grid assembly function. Individual hole-cutting facets (numbering around 60,000 as opposed to around 60 grids) were used as the basis for this decomposition. In order to reduce the memory and communication requirements, the result of each facet's hole-cutting operation was stored in a single array that was placed in shared memory using standard POSIX system calls [17]. This technique resulted in a nearly ideal load balance for the hole-cutting function.

This work extends this technique to the stencil search, which represents the other major piece of the grid assembly work. Here, the points that require interpolation form the basis for the decomposition of the work of the stencil search.

Grid Assembly

The grid assembly algorithms developed and implemented in the Beggar code [18] are being addressed in this parallelization effort. This code was developed at Eglin Air Force Base to simulate store separation events being addressed by the Air Force SEEK EAGLE Office (AFSEO). Beggar uses blocked, patched, and overlapping structured grids in a framework that includes grid assembly, flow solution, force and moment calculation, and the integration of the rigid body six degrees-of-freedom (6DOF) equations of motion.

Two major steps are involved in the grid assembly function: hole cutting and the stencil search. Holes are cut into overlapping grids to mark cells that intersect solid boundaries (rather than conforming to them) as invalid. This creates hole boundaries that require some sort of applied boundary condition. The points on the fringe of these hole boundaries and the points on the outer boundaries of any embedded grids are collectively referred to as intergrid boundary points (IGBPs). Boundary conditions are supplied at the IGBPs by interpolating the flow solution from any overlapping grids.

The *superblock* construct is introduced to aid in organizing the grid assembly function. The superblock is a collection of nonoverlapping grids that are treated as a single entity. Block-to-block connections are allowed only within a superblock; thus, a superblock is often used to implement a multiblock system of grids for part of the solution domain. Overlapping connections are allowed only between different superblocks.

Polygonal Mapping Tree

In order to establish overlapped grid communications, the following questions must be answered: does this point lie inside a grid, and if so, what is an appropriate interpolation stencil? These questions represent point-volume geometric relationships. In order to determine such relationships, Beggar uses a *polygonal mapping* (PM) tree, which is a combination of the *octree* and *binary space partioning* (BSP) tree data structures [19, 20].

An octree is a data structure in which a region of space is recursively subdivided into *octants*. Each *parent* octant is divided into eight *children* that can be further subdivided. This forms a hierarchy of *ancestor* and *descendant* octants. Each octant in the tree is termed a *node* with the

beginning node (lowest level) being the *root node* and the most descendent nodes (highest levels) being the *leaf nodes*. Such a data structure allows a domain to be divided into 8^n subdomains using just n levels. Therefore, a point can be identified as lying within a particular octant out of 8^n octants by using at most n comparisons (if the tree is perfectly balanced).

The BSP tree is a binary tree data structure in which each node of the tree is represented by a plane definition. Each node has two children representing the *in* and *out* sides of the plane. For a faceted representation of a surface, each facet defines a plane that is inserted into the BSP tree. Using a given point, the BSP tree is traversed by comparing the point against a plane definition at each level to determine which branch to descend into. Once a leaf node is reached, the point is identified as being inside or outside of the faceted surface.

In theory, a BSP tree of the cell faces on the boundaries of a grid block could be used to determine whether a point is IN or OUT of that particular grid. However, the structure of the tree is dependent on the order in which facets are inserted, and it is not guaranteed to be well balanced. Therefore, Beggar uses a combination of the octree and BSP tree data structures. The octree, which stays well balanced, is used to quickly narrow down the region of space in which a point lies. If a point lies in a leaf node that contains an overlapping boundary grid surface, it must be compared with a BSP tree that is stored in that leaf node to determine its relationship to that boundary surface and therefore its relationship to the grid itself.

A PM tree data structure is built for each superblock. The octree is refined until each octant contains no more than one grid boundary point. This produces a regular division of space that adapts to grid boundaries and grid point density. The boundary cell faces of the grids are then used to define facets that are inserted into BSP trees stored at the leaf nodes of the octree. Since each grid boundary point is normally shared by four cell faces and each octant contains only one grid boundary point, the BSP trees stored at the octree leaf nodes should be very shallow.

Once the basic data structure is complete, all of the octants of the leaf nodes are classified relative to the grid boundaries. Each octant is classified as inside or outside of each superblock or as a *boundary* octant. Then points can be classified efficiently relative to the superblocks.

Stencil Searching

The primary function of the PM tree is to help find interpolation stencils for grid points that require interpolated flow field information. When an interpolation stencil is required, the PM tree is used to classify the corresponding grid point relative to each superblock. This quickly identifies which superblocks the grid point lies in and therefore which superblocks might offer a source of interpolated information. However, once a point is identified as being inside a given superblock, the cell that contains the point must be found.

The algorithm used to perform this function is commonly referred to as *stencil jumping*. This algorithm uses Newton's method to invert the grid mapping from curvilinear space to Cartesian space. The end result of stencil jumping is the curvilinear coordinates that correspond to the Cartesian coordinates of a point that lies within a grid. Newton's method needs a good starting point; therefore, stored in the leaf nodes of the octree and the BSP trees are curvilinear coordinates at which to start the search. Although the PM tree classifies a point relative to a superblock, a starting point identifies a particular cell within a particular grid of the superblock. If the octree is sufficiently refined, the starting point should be close enough to ensure convergence.

Hole Cutting

Beggar uses an outline and fill algorithm for cutting holes. The facets of the hole-cutting surface are used to create an outline of the hole, and then the hole is filled by a fast sweep through the grid. Two complementary algorithms are available for marking the hole outline. The first algorithm marks the cells surrounding the vertices of a hole-cutting facet as being on the hole side or the world side of the facet. Then the facet is recursively refined in order to ensure a complete outline of the hole. The second algorithm performs a recursive neighbor search to identify all of the cells that intersect the facet if it is determined that refinement of the facet is necessary. Both algorithms treat each of the hole-cutting facets independently and only mark cells that are near the hole-cutting surface. The only restriction on the hole-cutting surfaces is that they form a closed body.

Parallel Implementations

Beggar has gone through an evolutionary process of parallel-processing development. Belk and Strassburg [21] developed the first parallel implementation. The grid assembly was performed as an initialization step, and only the flow solver was executed in parallel. Thus, this implementation was only applicable to static problems.

Prewitt, Belk, and Shyy [10] presented the first parallel implementation of Beggar (referred to as Phase I) that was applicable to dynamic problems. This implementation used a single front-end (FE) process to perform the grid assembly in a serial fashion with respect to the parallel execution of the flow solver across multiple back-end (BE) processes. This required that proper communication be established between the flow solution function and the grid assembly function; however, this did not require any consideration of load balancing or partitioning of the grid assembly function.

Prewitt, Belk, and Shyy [10] showed an improvement in the parallel efficiency of this implementation by overlapping execution time of the grid assembly function with that of the flow solver. This was possible because of the Newton-Relaxation scheme [22] used in the flow solver. The discretized, linearized, governing equations are written in the form of Newton's method. Each step of the Newton's method is solved using symmetric Gauss-Seidel (SGS) iteration. The SGS iterations, or *inner iterations*, are performed on a grid-by-grid basis, while the Newton iterations, or *dt iterations*, are used to achieve first-order time accuracy and are performed on all grids in sequence. This procedure eliminates synchronization errors at blocked and overset boundaries by iteratively bringing all dependent variables up to the next time level. Greater details on the flow solver can be found in the literature (see Whitfield [23] and Belk [24] for example).

For time-accurate flow calculations with Beggar, running more than one dt iteration to eliminate synchronization errors between grids and to achieve time accuracy is normal. Each dt iteration produces an updated approximation to the flow solution at the next time-step. Forces and moments can be calculated after each dt iteration using this approximate solution. If the forces and moments calculated after the first dt iteration are a good approximation to the final forces and moments, these values can be forwarded to the grid assembly process. This allows the computation of the grid assembly to proceed during the computation of additional dt iterations. If the computation time for the flow solver is sufficiently large, the computational cost of the

grid assembly process will be hidden.

This implementation (referred to as Phase II) is depicted in the timing diagram in Figure 1. This diagram shows the major functions executed during one time-step of the solution process. The vertical lines through a function indicate that the function is spread across multiple processors.

Prewitt, Belk, and Shyy [15] distributed the work of grid assembly using data decomposition techniques. Since the PM tree is used to classify points relative to superblocks, superblocks were chosen as the basis for coarse-grain data decomposition. The superblocks are distributed across multiple FE processes in an effort to equally balance the grid assembly work load. The work load is divided among the processes by cutting holes only into the superblocks mapped to a given process, and only the stencils donating to these superblocks are identified.

For this implementation, all of the PM trees are duplicated on each FE process. Therefore, each FE process has knowledge of the space occupied by each superblock and can identify all of the interpolation sources available for the IGBPs in any of the superblocks. All of the interpolation stencils that donate to IGBPs in superblocks on each processor are identified, and the best interpolation source is chosen without any communication. The only communication requirement is the global distribution of the cell-state information so that hole points and IGBPs can be identified.

Figure 2 shows the timing diagram for this implementation (referred to as Phase III) with the coarse-grain decomposition of the grid assembly function. This is essentially the same as Figure 1 except that the grid assembly function is distributed across multiple processors. A dynamic load-balancing function has also been added. Since the work load is dynamic, the grid assembly function is monitored to judge the load balance, and dynamic load balancing is performed by moving ownership of the superblocks between processors. The monitoring required to judge the load balance is accomplished by measuring the execution time of the grid assembly using system calls. The load-balancing function is executed during the calculation of the initial dt iteration; therefore, its execution time is hidden and does not adversely impact code performance.

A relatively small number of superblocks limit the ability to achieve a good load balance of the grid assembly work load. The splitting of grids, which is used to help balance the load of the flow solver, does not introduce new superblocks and, therefore, is of no help in load balancing the grid assembly work load. Instead, a finer grain decomposition of the work of the grid assembly function must be used to ensure the possibility of load balancing the grid assembly work load on any large number of processors.

The use of a fine-grain decomposition of the work associated with grid assembly will allow better load balancing on larger processor counts. Therefore, overlapping of the grid assembly time and the flow solution time could be used, with the increased number of processors used to decrease the grid assembly time, so that it continues to be hidden. However, if the fine-grain decomposition allows for a good load balance without excess overhead, the execution model could revert back to each function being spread across all of the available processors as shown in Figure 3. This would allow complete time accurate updating of the grid assembly.

Using Shared Memory

The work of the grid assembly function is split between hole cutting and the search for interpolation stencils. The work of the hole-cutting function is associated with the number of hole-cutting facets. The work of the search for interpolation stencils is associated with the number of IGBPs that require interpolation. Therefore, a fine-grain decomposition of the grid assembly function may be based on the view that the smallest piece of work is a single hole-cutting facet or an IGBP. Load balancing is achieved by equally distributing the weighted total number of hole-cutting facets and IGBPs across the available FE processes. Each facet and each IGBP is independent of its neighbors; therefore, communication between neighboring facets or IGBPs is not required. The only area of concern is the access and updating of several data structures by multiple processes.

Each hole-cutting facet marks a part of the hole outline, independent of the other facets. The hole-cutting algorithm identifies a set of cells near the facet and compares the cell centers to the plane definition of the facet. Some cells will be marked as holes; others will be marked as world-side. Some cells may even be marked as both holes and world-side. This information is called *cell-state* information and is stored as an array of bit-flags associated with each grid. The world-side status is needed to cap off the holes during the hole-filling process, while the hole status is needed to get the hole-filling process started.

When hole cutting is done with two neighboring facets on separate processors, the two cell state arrays must be merged to define the complete hole outline before the hole filling can be done. This merge is accomplished by a bitwise-or of the cell state status bits. This is a reduction operation and involves combining separate cell state arrays from each process for every grid in the system. This can be a rather expensive operation. For this reason, the cell state arrays are stored in shared memory.

When using shared memory, some facility is often required to make sure that only one process changes a variable at a time. However, in the case of hole cutting, it doesn't matter which facet marks a cell as a hole or as world-side. No matter how many facets mark a cell as a hole or world-side, the cell state information is only a set of bit flags. How many processes set a bit does not matter, as long as it gets set (no one ever clears these flags during grid assembly). Likewise, the order in which processes set a cell's status bit is immaterial. Thus, the cell-state information is stored in shared memory, and no coordination between processes is needed to ensure that the cell-state information is constructed correctly.

The use of IGBPs for fine-grain decomposition of the stencil search requires access to the complete PM tree. Without access to the complete PM tree, each process will be limited as to which IGBPs it can process. However, the duplication of the PM tree across multiple processes would require a large amount of memory. Therefore, the storage of the PM tree in shared memory is needed for the fine-grain decomposition of the stencil search based on IGBPs.

The process of allocating shared memory with POSIX calls consists of three steps. First, a shared-memory segment must be opened with a call analogous to the call used to open a file. Then the size of the shared-memory segment must be specified. Finally, the shared-memory segment must be mapped to a physical address. The pointer to the shared-memory segment can then be used like any other pointer to memory.

The use of shared memory to store the cell-state array is straightforward. Since it is a simple array of integers, a shared-memory segment of the appropriate size is mapped and used as a

simple array. However, the PM tree is a linked-list data structure that contains pointers. This complicates the situation, because the pointers must be valid in relation to the shared-memory segment that is mapped.

It is not known a priori how many nodes will be required to hold the complete PM tree. Therefore, memory is dynamically allocated. A cache structure is used to reduce the number of system calls used to allocate memory. Splitting the work of building the PM tree across multiple processes and building the PM tree directly in shared memory would be desirable. However, with multiple processes mapping shared memory, different processes may map different segements of shared memory to the same physical address. This would make creating a valid set of pointers linking the complete data structure impossible. Therefore, the approach that has been taken is to create the data structure and then to copy it into shared memory. The pointers are corrected during the copy so that they point into the correct location of shared memory, and the mapping of the shared-memory segments is coordinated between the processes so as to ensure that they all map the shared memory in a consistent fashion.

Test Problem

The three store, ripple release case, first presented by Thoms and Jordan [25] and later modeled using Beggar by Prewitt, Belk, and Maple [26], is being used as a test case for judging parallel performance. The geometric configuration is that of three generic stores in a triple ejector rack (TER) configuration under a generic pylon that is attached to a clipped delta wing. This configuration and the history of the motion is depicted in Figure 4 with the stores under the right wing. The three generic stores are identical bodies of revolution with an ogive-cylinder-ogive planform shape and four clipped delta fins with a NACA 0008 airfoil cross section. The pylon is an extruded surface of similar ogive-cylinder-ogive cross section. The wing has a 45-degree leading edge sweep and a NACA 64A010 airfoil cross section.

Ejectors are used to help ensure a safe separation trajectory. Each store is acted on by a pair of ejectors that create a nose up pitching moment to counteract a strong aerodynamic nose down pitching moment seen in carriage. The ejectors are directed downward on the bottom store of the TER and are directed outward at 45 degrees (with respect to vertical) on the two shoulder stores. The stores are released in bottom-outboard-inboard order with a 0.04-sec delay between each release. The ejectors are applied at release for a duration of 0.045 sec.

The grid system consists of 10 superblocks, 67 blocks, and approximately 2.2 million grid points. The largest grid in this grid system has 62,370 grid points. This set of grids should extend the maximum processor count beyond 32.

Since the work of the flow solver is closely associated with the number of grid points, the load balance of the flow solver work can be judged by the distribution of the grid points. Table 1 lists the load imbalance factors achieved based solely on the number of grid points per processor. The third column lists the effective number of processors, which is equal to the actual number of processors divided by the load imbalance factor. As can be seen, the flow solver should load balance relatively well up to 40 processors. Beyond this point, an increase in the processor count is offset by the load imbalance.

The flight conditions simulated in all of the test runs are for a freestream Mach number of 0.95 at 26,000 feet altitude. All solutions are calculated assuming inviscid flow. The flow solver

is run time-accurately for 600 time-steps of 0.0005 sec giving a total of 0.3 sec of the trajectory.

A single-processor simulation was computed on an SGI Origin 2000 with 195 MHz R10000 processors and 2 processors and 512 MB of memory per node. This baseline solution time was 9,384 min (about 6.5 days). The execution time of the grid assembly function was compared with the total execution time from the sequential run to establish that 5 percent of the work is associated with the grid assembly function, while 95 percent of the work is associated with the flow solver.

Results

The ripple release test problem was run using the superblocks as the basis for decomposition of the grid assembly (Phase III). The timing results are presented in Figure 5. Some of the curves are labeled with F_i and G_i, which are the load imbalance factors for the flow solver and the grid assembly, respectively. All of the runs used 4 FE processes, and the number of BE processes varied between 16 and 40. Dynamic load balancing of the grid assembly function was performed after each time-step. The speedup experienced followed the estimated speedup curve, which is based on Amdahl's Law (see reference [15] for the derivation), up to 24+4 (24 BE and 4 FE) processes; however, for larger processor counts, the grid assembly time failed to be hidden, and the speedup fell well below the estimated value.

Based on this data, the load imbalance of the grid assembly function is judged to be about 65 percent. This is the imbalance of the grid assembly as a whole. To better judge the load balance of the hole cutting and stencil search routines, the execution times of these two routines are plotted in Figures 6 and 7. Each curve in the two plots represents a different FE process. The grouping or separation between the curves represents the variation in the execution times on the different processes; thus, it indicates the quality of the load balance. From these plots, one can see that the stencil search routine (Figure 7) is much better balanced than the hole-cutting routine (Figure 6). The stencil search imbalance is always less than 10 percent, while the hole cutting imbalance is in the range of 50 to 70 percent.

Since hole cutting is the more imbalanced of the two major components of the grid assembly, the hole-cutting process was the first to be decomposed using a fine-grain treatement. It was also the easier to implement, using shared memory to store the cell-state information only. The hole-cutting facets are divided between the FE processes, and each FE process cuts holes into all of the superblocks with all of the facets that are mapped to that process. The execution time of the hole-cutting function on each FE process is used as a weight for dynamically adjusting the number of facets that are mapped to each process.

Figures 8 and 9 show the execution times of the hole-cutting function when 4 or 8 FE processes are used in combination with dynamic load balancing of the hole cutting. These figures may appear to contain only a single curve; however, each plot actually contains one curve for each FE process. The tight grouping of the curves indicates the remarkable load balance that was achieved.

It takes several time steps at the beginning of a run for the execution times to converge, and small changes in the execution of the computer, which can cause variations in the execution time of a process, can cause perturbations in the load balance. The load balance quickly recovers from these; however, some sort of limiting or damping of the changes between the time steps

may help to maintain the load balance.

Figure 10 shows the improvement in performance for the entire simulation when the fine-grain treatment of the hole cutting was used. The reduction in the execution time of the grid assembly was sufficient to allow the grid assembly time to be hidden behind the execution time of the flow solver on the 28+4 process run. The 32+4 and 36+4 process run did not show significant improvements, but the 40+4 process run performed well.

The overall performance is still significantly less than what might be assumed from the 10 percent imbalance of the stencil search as seen in Figure 7. One probable cause for this is the fact that the stencil search is an iterative process that includes synchronization between iterations. Such synchronization within an iterative loop can cause additional load imbalance if each iteration of the loop is not well balanced.

A fine-grain treatment of the search for interpolation stencils has been implemented using the number of IGBPs as the basis. In comparison to the fine-grain implementation of the hole-cutting function, this implementation is more complicated because the number of IGBPs is not constant and the stencil search is an iterative function.

With each iteration of the stencil search, some IGBPs may fail to be interpolated and will be marked as OUT. This causes additional points to become IGBPs on the next iteration of the stencil search. Thus, the number of new IGBPs has to be counted with each iteration.

This also complicates the algorithm that was used for dynamic load balancing of the fine-grain hole cutting. In this algorithm, the execution time from the previous time-step is used to modify the number of pieces of work that are mapped to an FE process. But, with the stencil search, the execution times vary with each iteration of the stencil search because of significant changes in the number of new IGBPs. Thus, dynamic load balancing of the fine-grain stencil search is not yet being done. Instead, the new IGBPs are being divided equally across the FE processes.

Figure 11 shows the execution times of the stencil search when 4 FE processes are used. The PM tree was not stored in shared memory, but was duplicated across the FE processes. This run was executed on an Origin 3800 (about twice as fast as the Origin 2000 used in the previous runs); therefore, this plot should not be compared directly to the previous plots, but it can be used to judge the load balance. Because of the lack of dynamic load balancing, the load balance is not as good as that seen with the fine-grain hole cutting (with dynamic load balancing). However, the load balance is slightly better than that shown in Figure 7, and the fine-grain decomposition will greatly increase the scalability.

The tests required to show the overall improvement in performance have not yet been completed. However, the improvement in scalability because of the fine-grain treatment of the hole cutting and the stencil search should make possible the Phase IV implementation with an acceptable load balance.

Conclusions

Improvements in the parallel implementation of the grid assembly function within the Beggar code based on a mixed programming model were presented. The programming model combines message passing and shared-memory constructs based on the use of standard POSIX system calls to allocated shared memory. The shared memory was used to store large, common data

structures including cell-state information that is used in the hole-cutting function and the PM trees used in stencil searching. These data structures ranged from one-dimensional arrays to dynamically allocated link-lists. The use of shared memory facilitates the use of a fine-grain decomposition of the work associated with the grid assembly process based on the hole-cutting facets and the IGBPs. In combination with dynamic load balancing, the load balance of the grid assembly and the overall simulation performance were shown to be greatly improved for the example problem addressed.

Although the use of shared memory reduces the total memory required, if a large number of processors in a non-uniform memory access (NUMA) architecture try to access a single copy of a data structure, performance can be degraded by memory bandwidth limitations. In order to avoid this problem, multiple copies of the PM tree, which are shared by a subset of the total processes, may be required. Such an arrangement is a subject for future experimentation.

Acknowledgments

This work was performed under the collaborative support of the Air Force Seek Eagle Office. Support was provided in part by a grant of computer time from the DoD High Performance Computing Modernization Program at the Aeronautical Systems Center Major Shared Resource Center and the U.S. Army Engineer Research and Development Center Major Shared Resource Center. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Benek, J.A., Steger, J.L., and Dougherty, F.C., "A Flexible Grid Embedding Technique with Applications to the Euler Equations," AIAA Paper 83-1944, June 1983.

[2] Lijewski, L.E., and Suhs, N., "Time-Accurate Computational Fluid Dynamics Approach to Transonic Store Separation Trajectory Prediction," *Journal of Aircraft*, Vol. 31, No. 4, pp. 886-891, August 1994.

[3] Meakin, R.L., "Moving Body Overset Grid Methods for Complete Aircraft Tiltrotor Simulations," AIAA Paper 93-3350-CP, AIAA Computational Fluid Dynamics Conference, July 1993, pp. 576-588.

[4] Suhs, N.E., and Tramel, R.W., "PEGSUS 4.0 USER'S MANUAL," AEDC-TR-91-8, June 1991.

[5] Meakin, R.L., "A New Method for Establishing Intergrid Communication among Systems of Overset Grids," AIAA Paper 91-1586, June 1991.

[6] Chesshire, G., and Henshaw, W.D., "Composite Overlapping Meshes for the Solution of Partial Differential Equations," *Journal of Computational Physics*, Vol. 90, No. 1, September 1990.

[7] Maple, R.C., and Belk, D.M., "Automated Set Up of Blocked, Patched, and Embedded Grids in the Beggar Flow Solver," *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, ed. N.P. Weatherill et al., Pine Ridge Press, pp. 305-314, 1994.

[8] Chiu, Ing-Tsau, and Meakin, R.L, "On Automating Domain Connectivity for Overset Grids," NASA-CR-199522, August 1995.

[9] Chesshire, G., Brislawn, D., Brown, D., Henshaw, W., Quinlan, D., and Saltzman, J., "Efficient Computation of Overlap for Interpolation between Moving Component Grids," 3rd Symposium on Overset Composite Grid and Solution Technology, November 1996.

[10] Prewitt, N.C., Belk, D.M., and Shyy, Wei, "Implementations of Parallel Grid Assembly for Moving Body Problems," AIAA Paper 98-4344, August 1998.

[11] Barszcz, E., Weeratunga, S.K., and Meakin, R.L., "Dynamic Overset Grid Communication on Distributed Memory Parallel Processors," AIAA Paper 93-3311, July 1993.

[12] Wissink, A.M., and Meakin, R.L., "On Parallel Implementations of Dynamic Overset Grid Methods," SC97: High Performance Networking and Computing, November 1997.

[13] Wissink, A.M., and Meakin, R.L., "Computational Fluid Dynamics with Adaptive Overset Grids on Parallel and Distributed Computer Platforms," International Conference on Parallel and Distributed Computing, July 1998.

[14] Meakin, R.L., and Wissink, A.M., "Unsteady Aerodynamic Simulation of Static and Moving Bodies Using Scalable Computers," AIAA Paper 99-3302, July 1999.

[15] Prewitt, N.C., Belk, D.M., and Shyy, W., "Distribution of Work and Data for Parallel Grid Assembly," AIAA Paper 99-0913, January 1999.

[16] Prewitt, N.C., "Parallel Computing of Overset Grids for Aerodynamic Problems With Moving Objects," PhD Dissertation, University of Florida, December 1999.

[17] Gallmeister, B.O., *POSIX.4: Programming for the Real World*, O'Reilly & Associates, Sebastopol, CA, 1995.

[18] Belk, D.M., and Maple, R.C., "Automated Assembly of Structured Grids For Moving Body Problems," AIAA Paper 95-1680, June 1995.

[19] Samet, H., *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[20] Samet, H., *Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[21] Belk, D.M., and Strasburg, D.W., "Parallel Flow Solution on the T3D with Blocked and Overset Grids," 3rd Symposium on Overset Composite Grid and Solution Technology, November 1996.

[22] Whitfield, D.L., "Newton-Relaxation Schemes for Nonlinear Hyperbolic Systems," Engineering and Industrial Research Station Report MSSU-EIRS-ASE-90-3, Mississippi State University, October 1990.

[23] Whitfield, D., and Taylor, L., "Discretized Newton-Relaxation Solution of High Resolution Flux-Difference Split Schemes," AIAA Paper 91-1539, June 1991.

[24] Belk, D.M. *Unsteady Three-Dimensional Euler Equations Solutions on Dynamic Blocked Grids*, PhD Dissertation, Mississippi State University, August 1986.

[25] Thoms, R.D., and Jordan, J.K., "Investigations of Multiple Body Trajectory Prediction Using Time Accurate Computational Fluid Dynamics," AIAA Paper 95-1870, June 1995.

[26] Prewitt, N.C., Belk, D.M., and Maple, R.C., "Multiple Body Trajectory Calculations Using the Beggar Code," AIAA Paper 96-3384, July 1996.

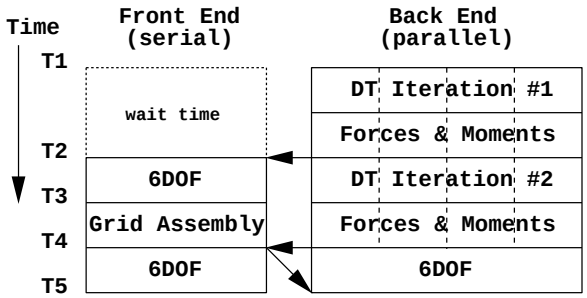

Figure 1: Function overlapping implementation

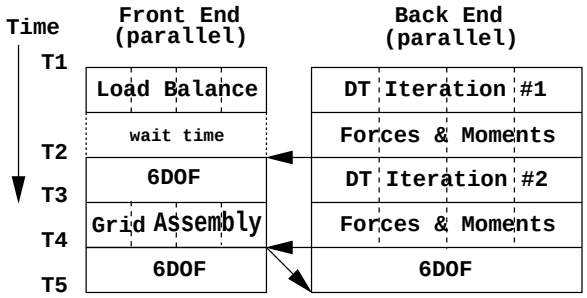

Figure 2: Coarse-grain decomposition of grid assembly based on superblocks

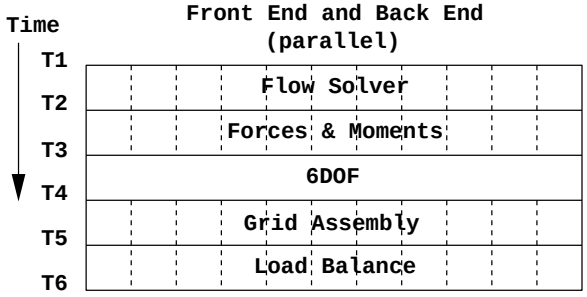

Figure 3: Fine-grain decomposition of grid assembly

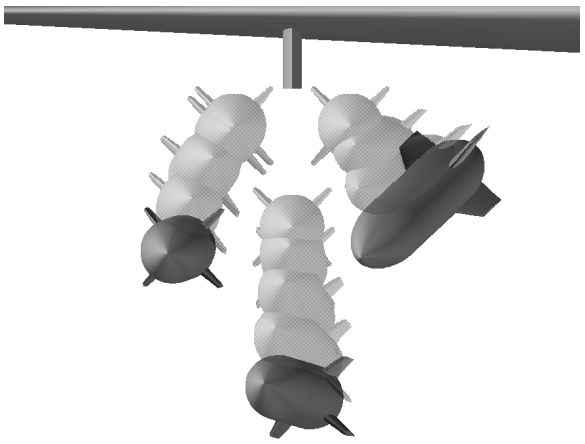

Figure 4: History of the motion of the three store separation

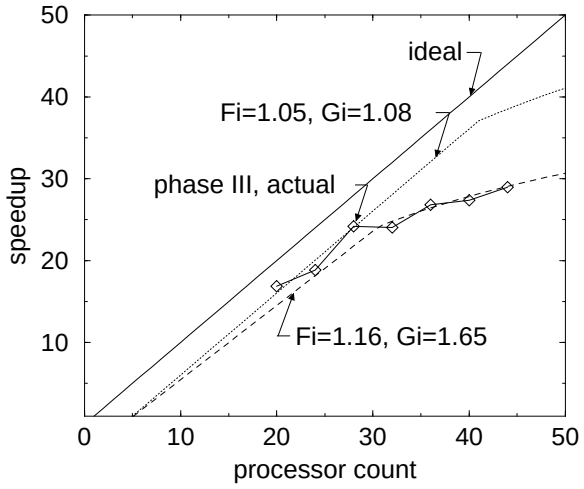

Figure 5: Speedup of Phase III

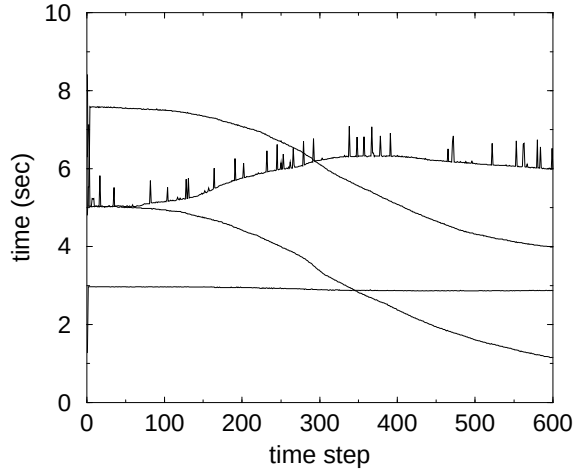

Figure 6: Execution times of hole cutting with load balancing based on the measured execution times of stencil searching (each curve represents a separate process)

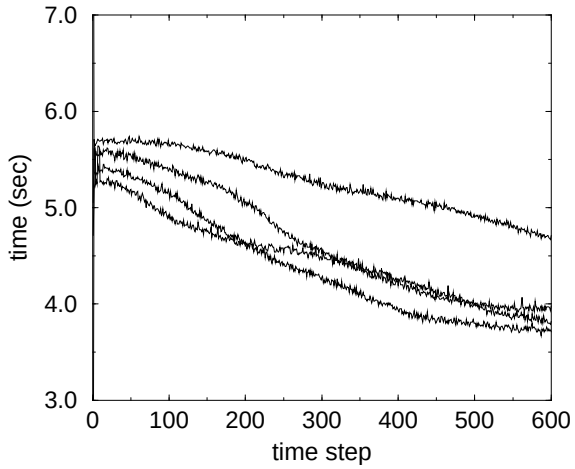

Figure 7: Execution times of stencil searching with load balancing based on the measured execution time of stencil searching (each curve represents a separate process)

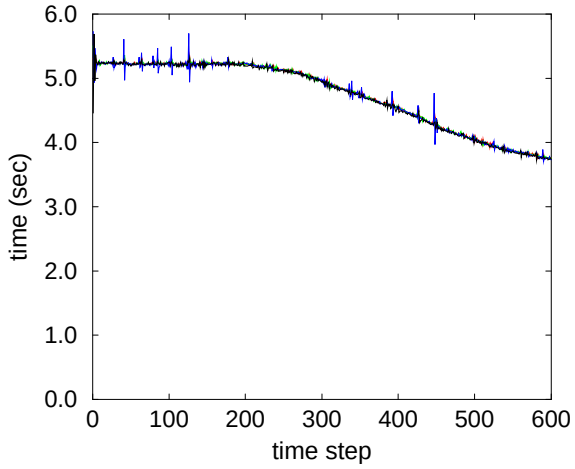

Figure 8: Execution times for dynamically load balanced, fine-grain hole cutting distributed across 4 FE processes

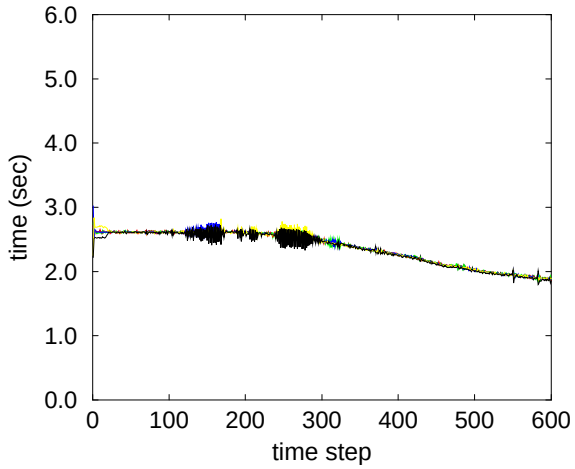

Figure 9: Execution times for dynamically load balanced, fine-grain hole cutting distributed across 8 FE processes

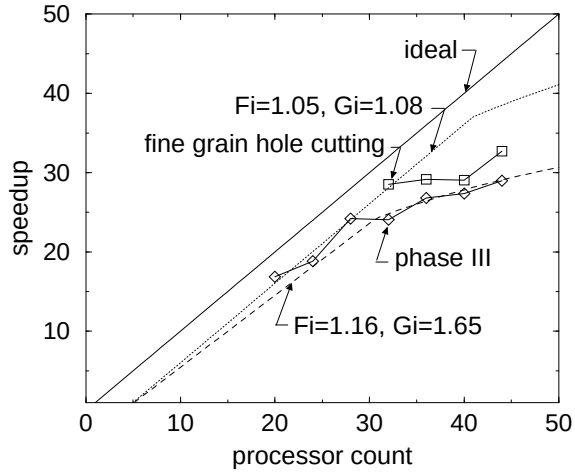

Figure 10: Comparison of speedup including fine-grain hole cutting with dynamic load balancing

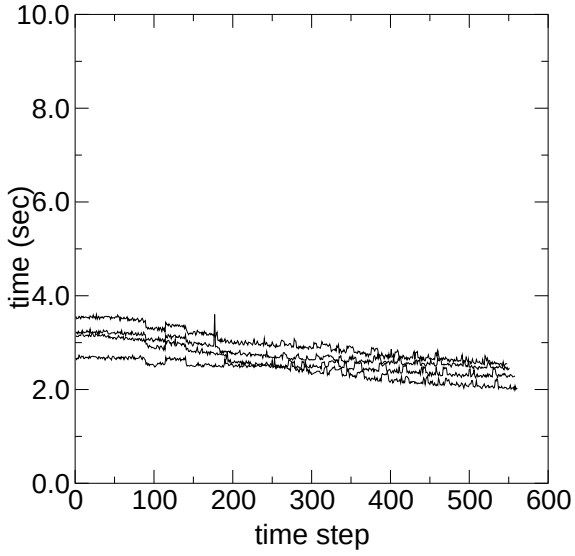

Figure 11: Execution times for fine-grain stencil searching distributed across 4 FE processes

Table 1: Load imbalance factors

No. of Processors	F_i	Effec. No. of Processors
4	1.005	3.98
8	1.04	7.69
12	1.05	11.43
16	1.05	15.24
20	1.07	18.69
24	1.06	22.64
28	1.14	24.56
32	1.07	29.91
36	1.18	30.51
40	1.12	35.71
44	1.23	35.77
48	1.35	35.56